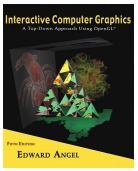


Name: _____ Term # ____

Homework 2 SOLUTIONS

(400 points)

NOTE: **Chapter 2 of the textbook shows the building blocks of graphics programming.**
Part A is intended to be done by hand.
Part B is an openGL application.

Please note that this Homework is worth 400 points.

A. (300 pts) Paper and Pencil

(Guidelines: *Read the material from the textbook chapter, you can use textbook figures to exemplify your answer, use keywords, summarize your answer, but the answer* ***cannot be longer the 7 lines!****)*

2.1 THE SIERPINSKI GASKET

We use as a sample problem the drawing of the Sierpinski gasket—an interesting shape that has a long history and is of interest in areas such as fractal geometry.

a. Describe the construction of the 2D Sierpinski gasket.

ANSWER:

(Angel pp. 40)

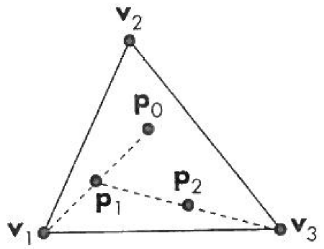


FIGURE 2.1 Generation of the Sierpinski gasket.

1. Pick an initial point (x, y, z) at random inside the triangle.
2. Select one of the three vertices at random.
3. Find the location halfway between the initial point and the randomly selected vertex.
4. Display this new point by putting some sort of marker, such as a small circle, at the corresponding location on the display.
5. Replace the point at (x, y, z) with this new point.
6. Return to step 2.

2.2 PROGRAMMING TWO-DIMENSIONAL APPLICATIONS

We use as a sample problem the drawing of the Sierpinski gasket—an interesting shape that has a long history and is of interest in areas such as fractal geometry.

a. Describe the representation of a three-dimensional point **p.**

ANSWER:

(Angel pp. 41)

We can implement representations of points in a number of ways, but the simplest is to think of a three-dimensional point as being represented by a triplet $\mathbf{p} = (x, y, z)$ or a column matrix

$$\mathbf{p} = \begin{bmatrix} x \\ y \\ z \end{bmatrix},$$

whose components give the location of the point. For the moment, we can leave aside the question of the coordinate system in which p is represented.

b. Define the vertex.

ANSWER:

(Angel pp. 41)

We use the terms *vertex* and *point* in a somewhat different manner in OpenGL. A **vertex** is a position in space; we use two-, three-, and four-dimensional spaces in computer graphics. We use vertices to define the atomic geometric primitives that are recognized by our graphics system. The simplest geometric primitive is a point in space, which is specified by a single vertex. Two vertices define a line segment, a second primitive object; three vertices can determine either a triangle or a circle; four vertices determine a quadrilateral, and so on.

OpenGL has multiple forms for many functions. The variety of forms allows the user to select the one best suited for her problem. For the vertex functions, we can write the general form

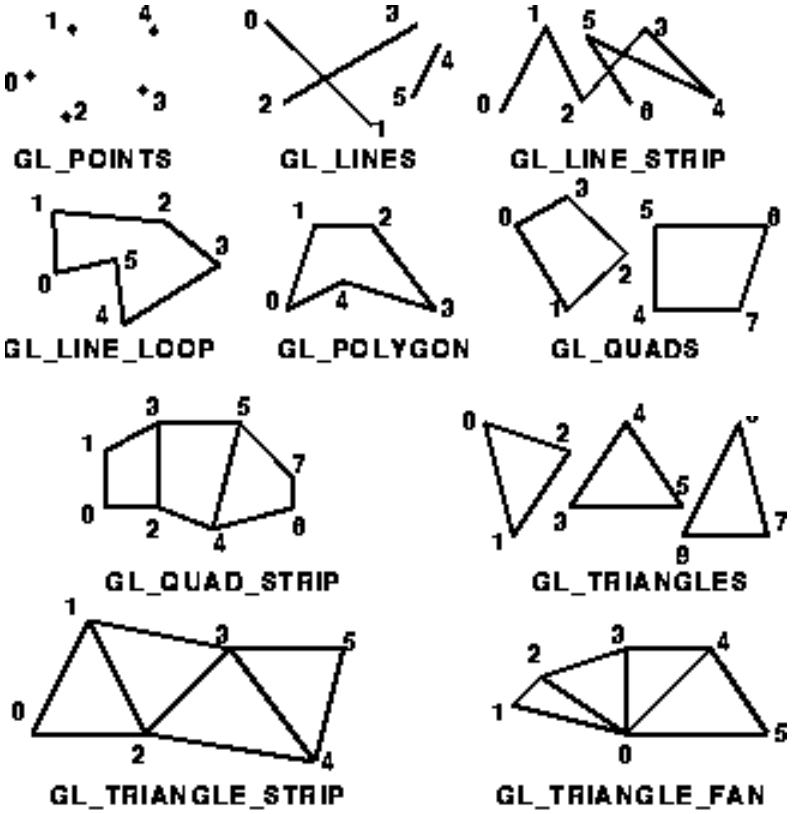
```
glVertex*()
```

where the * can be interpreted as either two or three characters of the form `nt` or `ntv`, where `n` signifies the number of dimensions (2, 3, or 4); `t` denotes the data type, such as integer (`i`), float (`f`), or double (`d`); and `v`, if present, indicates that the variables are specified through a pointer to an array, rather than through an argument list. We will use whatever form is best suited for our discussion, leaving the details of the various other forms to the *OpenGL Reference Manual* [Ope04]. Regardless of which form a user chooses, the underlying representation is the same, just as the plane on which we are constructing the gasket can be looked at as either a two-dimensional space or the subspace of a three-dimensional space corresponding to the plane $z = 0$. In Chapter 4, we will see that the underlying representation is four-dimensional; however, we do not need to worry about that fact yet.

c. Describe all the openGL geometric primitives that can be built with vertices.

ANSWER:

(Angel pp. 42)



The call to `glFlush` ensures that points are rendered to the screen as soon as possible. If you leave it out, then the program should work correctly, but you may notice a delay in a busy or networked environment. We still do not have a complete

2.2.1 Coordinate Systems

At this point, you may be puzzled about how to interpret the values of `x`, `y`, and `z` in our specification of vertices. In what units are they? Are they in feet, meters, microns? Where is the origin? In each case, the simple answer is that it is up to you.

a. Describe the world coordinate system and screen coordinate system.

ANSWER:

(Angel pp. 46)

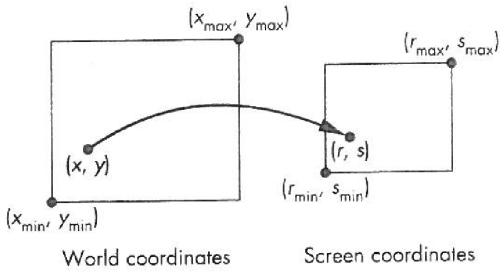


FIGURE 2.3 Mapping from world coordinates to screen coordinates.

Originally, graphics systems required the user to specify all information, such as vertex locations, directly in units of the display device. If that were true for high-level application programs, we would have to talk about points in terms of screen locations in pixels or centimeters from a corner of the display. There are obvious problems with this method, not the least of which is the absurdity of using distances on the computer screen to describe phenomena where the natural unit might be light years (such as in displaying astronomical data) or microns (for integrated-circuit design). One of the major advances in graphics software systems occurred when the graphics systems allowed users to work in any coordinate system that they desired. The advent of **device-independent graphics** freed application programmers from worrying about the details of input and output devices. The user's coordinate system became known

as the **world coordinate system**, or the **application**, **model**, or **object coordinate system**. Within the slight limitations of floating-point arithmetic on our computers, we can use any numbers that fit our application.

Units on the display were first called **physical-device coordinates** or just **device coordinates**. For raster devices, such as most CRT displays, we use the term **window coordinates** or **screen coordinates**.[3] Screen coordinates are always expressed in some integer type, because the center of any pixel in the frame buffer must be located on a fixed grid or, equivalently, because pixels are inherently discrete and we specify their locations using integers.

At some point, the values in world coordinates must be mapped to screen coordinates, as shown in Figure 2.3. The graphics system, rather than the user, is responsible for this task, and the mapping is performed automatically as part of the rendering process. As we will see in the next few sections, to define this mapping the user needs to specify only a few parameters—such as the area of the world that she would like to see and the size of the display.

2.3 THE OPENGL API

2.3.1 Graphics Functions

We have the heart of a simple graphics program; now, we want to gain control over how our objects appear on the display. We also want to control the flow of the program, and we have to interact with the window system. Befc

FIGURE 2.4 Graphics system as a black box.

a. Classify openGL APIs and briefly describe each group.

ANSWER:

(Angel pp. 47)

We describe an API through the functions in its library. A good API may contain hundreds of functions, so it is helpful to divide them into seven major groups:

1. Primitive functions
2. Attribute functions
3. Viewing functions
4. Transformation functions
5. Input functions
6. Control functions
7. Query functions

The **primitive functions** define the low-level objects or atomic entities that our system can display. Depending on the API, the primitives can include points, line segments, polygons, pixels, text, and various types of curves and surfaces.

If primitives are the *what* of an API—the primitive objects that can be displayed—then attributes are the *how*. That is, the attributes govern the way that a primitive appears on the display. **Attribute functions** allow us to perform operations ranging from choosing the color with which we display a line segment, to picking a pattern with which to fill the inside of a polygon, to selecting a typeface for the titles on a graph.

Our synthetic camera must be described if we are to create an image. As we saw in Chapter 1, we must describe the camera's position and orientation in our world and must select the equivalent of a lens. This process will not only fix the view but also allow us to clip out objects that are too close or too far away. The **viewing functions** allow us to specify various views, although APIs differ in the degree of flexibility they provide in choosing a view.

One of the characteristics of a good API is that it provides the user with a set of **transformation functions** that allows her to carry out transformations of objects, such as rotation, translation, and scaling. Our developments of viewing in Chapter 5 and of modeling in Chapter 10 will make heavy use of matrix transformations. In OpenGL, we obtain viewing and modeling functionality through a small set of transformations functions.

For interactive applications, an API must provide a set of **input functions** to allow us to deal with the diverse forms of input that characterize modern graphics systems. We need functions to deal with devices such as keyboards, mice, and data tablets. In Chapter 3, we will introduce functions for working with different input modes and with a variety of input devices.

In any real application, we also have to worry about handling the complexities of working in a multiprocessing, multiwindow environment—usually an environment where we are connected to a network and there are other users. The **control functions** enable us to communicate with the window system, to initialize our programs, and to deal with any errors that take place during the execution of our programs.

If we are to write device-independent programs, we should expect the implementation of the API to take care of differences between devices, such as how many colors are supported or the size of the display. However, there are applications where we need to know some properties of the particular imp
would probably choose to do things differently if we knew in advance that we were working with a display that could support only two colors rather than millions of colors. More generally, within our applications we can often use other information within the API, including camera parameters or values in the frame buffer. A good API provides this information through a set of **query functions**.

2.3.2 The Graphics Pipeline and State Machines

If we put together some of these perspectives on graphics APIs, we can obtain another view, one closer to the way OpenGL, in particular, is actually organized and implemented. We can think of the entire graphics system as a **state machine**, a black box that contains a finite-state machine. This state machine has inputs that come from the application program. These inputs may change the state of the machine or can cause the machine to produce a visible output. From the perspective of the API, graphics functions are of two types: those that define primitives that flow through a pipeline inside the state machine and those that either change the state inside the machine or return state information. In OpenGL, functions such as `glVertex` are of the first type, whereas almost all other functions are of the second type.

a. Describe one parameter that is maintained in the openGL state.

ANSWER:

(Angel pp. 49)

One important consequence of this view is that in OpenGL most parameters are persistent; their values remain unchanged until we explicitly change them through functions that alter the state. For example, once we set a color, that color remains the *current color* until it is changed through a color-altering function. Another consequence of this view is that attributes that we may conceptualiz
a red line or a blue circle—are in fact part of the state, and a line will be drawn in red only if the current color state calls for drawing in red. Although within our applications it is usually harmless, and often preferable, to think of attributes as bound to primitives, there can be annoying side effects if we neglect to make state changes when needed or lose track of the current state.

2.4 PRIMITIVES AND ATTRIBUTES

Within the graphics community, there has been an ongoing debate about which primitives should be supported in an API. The debate is an old one and has never been fully resolved. On the minimalist side, the contention is that an API should contain a small set of primitives that all hardware can be expected to support. In addition, the primitives should be orthogonal, each giving a capability unobtainable from the others. Minimal systems typically support lines, polygons, and some form of text (strings of characters), all of which can be generated efficiently in hardware. On the other end are systems that can also support a variety of primitives, such as circles, curves, surfaces, and solids. The argument here is that users need more complex primitives to build sophisticated applications easily. However, because few hardware systems can be expected to support the large set of primitives that is the union of all the desires of the user community, a program developed with such a system probably would not be portable, because few implementations could be expected to support the entire set of primitives.

OpenGL takes an intermediate position. The basic library has a small set of primitives. The GLU library contains a richer set of objects derived from the basic library.

a. Describe the two classes of primitives in openGL.

ANSWER:

(Angel pp. 50)

OpenGL supports two classes of primitives: **geometric primitives** and **image, or raster, primitives**. Geometric primitives are specified in the problem domain and include points, line segments, polygons, curves, and surfaces. These primitives pass

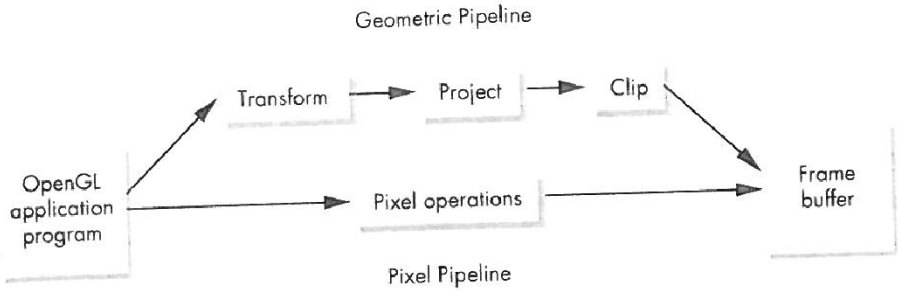


FIGURE 2.6 Simplified OpenGL pipeline.

through a geometric pipeline, as shown in Figure 2.6, where they are subject to a series of geometric operations that determine whether a primitive is visible, where on the display it appears if it is visible, and the rasterization of the primitive into pixels in the frame buffer. Because geometric primitives exist in a tv
space, they can be manipulated by operations such as rot
addition, they can be used as building blocks for other geometric objects using these same operations. Raster primitives, such as arrays of pixels, lack geometric properties and cannot be manipulated in space in the same way as geometric primitiv
pass through a separ
their way to the frame buffer.

2.4.1 Polygon Basics

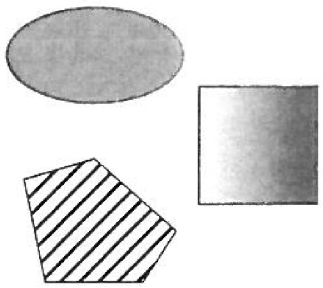


FIGURE 2.8 Filled objects.

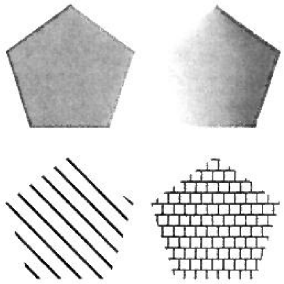


FIGURE 2.9 Methods of displaying a polygon.

Line segments and polylines can model the edges of objects, but closed objects also may have interiors (Figure 2.8). Usually we reserve the name **polygon** for an object that has a border that can be described by a line loop but also has a well-defined interior.[4] Polygons play a special role in computer graphics because we can display them rapidly and use them to approximate arbitrary surfaces. The performance of graphics systems is characterized by the number of polygons per second that can be rendered.[5] We can render a polygon in a variety of ways: We can render only its edges; we can render its interior with a solid color or a pattern; and we can render or not render the edges, as shown in Figure 2.9. Although the outer edges of a polygon are defined easily by an ordered list of vertices, if the interior is not well defined, then the list of vertices may not be rendered at all or rendered in an undesirable manner. Three properties will ensure that a polygon will be displayed correctly: It must be simple, convex, and flat.

a. Describe a convex polygon.

ANSWER:

(Angel pp. 53)

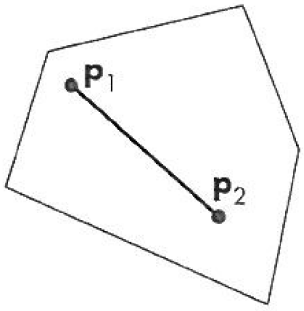


FIGURE 2.11 Convexity.

From the perspective of implementing a practical algorithm to fill the interior of a polygon, simplicity alone is often not enough. Some APIs guarantee a consistent fill from implementation to implementation only if the polygon is convex. An object is **convex** if all points on the line segment between any two points inside the object, or on its boundary, are inside the object. Thus, in Figure 2.11, $\mathbf{p}_1$ and $\mathbf{p}_2$ are arbitrary points inside a polygon and the entire line segment connecting them is inside the polygon. Although so far we have been dealing with only two-dimensional objects, this definition makes reference neither to the type of object nor to the number of dimensions. Convex objects include triangles, tetrahedra, rectangles, circles, spheres, and parallelepipeds (Figure 2.12). There are various tests for convexity (see Exercise 2.19). However, like simplicity testing, convexity testing is expensive and usually left to the application program.

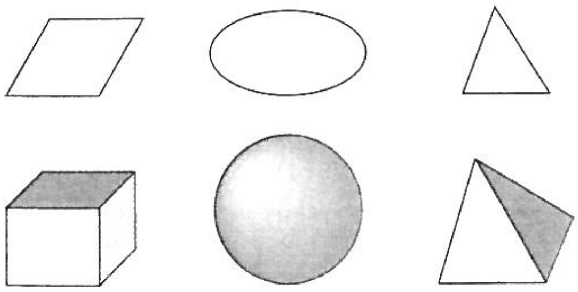


FIGURE 2.12 Convex objects.

2.4.3 Approximating a Sphere

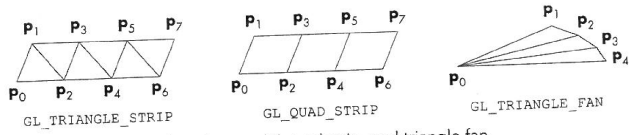


FIGURE 2.14 Triangle strip, quadrilateral strip, and triangle fan.

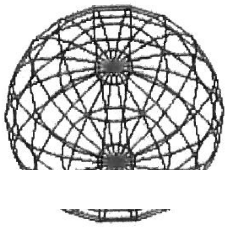


FIGURE 2.15 Sphere approximation with quadrilaterals.

Fans and strips allow us to approximate many curved surfaces simply. For example, one way to construct an approximation to a sphere is to use a set of polygons defined by lines of longitude and latitude as shown in Figure 2.15. We can do so very efficiently using either quad strips or triangle strips. Consider a unit sphere. We can describe it by the following three equations:

a. Write and explain the three equations and explain how the quadrilaterals are used to approximate the unit sphere.

ANSWER:

(Angel pp. 55)

Applet:
http://www.flashandmath.com/mathlets/multicalc/coords/shilmay23fin.html

$$x(\theta, \phi) = \sin\theta \cos\phi,$$

$$y(\theta, \phi) = \cos\theta \cos\phi,$$

$$z(\theta, \phi) = \sin\phi.$$

If we fix θ and draw curves as we change ϕ, we get circles of constant longitude. Likewise, if we fix ϕ and vary θ, we obtain circles of constant latitude. By generating points at fixed increments of θ and ϕ, we can define quadrilaterals as shown in Figure 2.15. Remembering that we must convert degrees to radians for the standard trigonometric functions, the code for the quadrilaterals corresponding to increments of 20 degrees in θ and to 20 degrees in ϕ is

```
for(phi=-80.0; phi<=80.0; phi+=20.0)
{
    phir=c*phi;
    phir20=c*(phi+20);
    glBegin(GL_QUAD_STRIP);
    for(theta=-180.0; theta<=180.0;theta+=20.0)
    {
        thetar=c*theta;
        x=sin(thetar)*cos(phir);
        y=cos(thetar)*cos(phir);
        z=sin(phir);
        glVertex3d(x,y,z);
        x=sin(thetar)*cos(phir20);
        y=cos(thetar)*cos(phir20);
        z=sin(phir20);
        glVertex3d(x,y,z);
    }
    glEnd();
}
```

However, we have a problem at the poles, where we can no longer use strips because all lines of longitude converge there. We can, however, use two triangle fans, one at each pole as follows:

```
glBegin(GL_TRIANGLE_FAN);
   glVertex3d(0.0, 0.0 , 1.0);
   c=M_PI/180.0;
   c80=c*80.0;
```

2.4.4 Text

Graphical output in applications such as data analysis and display requires annotation, such as labels on graphs. Although, in nongraphical programs, textual output is the norm, text in computer graphics is problematic. In nongraphical applications, we are usually content with a simple set of characters, always displayed in the same manner. In computer graphics, however, we often wish to display text in a multitude of fashions by controlling type styles, sizes, colors, and other parameters. We also want to have available a choice of fonts. **Fonts** are families of typefaces of a particular style, such as Times, Computer Modern, or Helvetica.

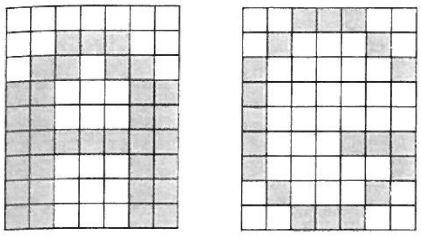


FIGURE 2.17 Raster text.

a. What is the openGL function for text ?

ANSWER:

(Angel pp. 57)

Because stroke and bitmap characters can be created from other primitives, OpenGL does not have a text primitive. However, the GLUT library provides a few predefined bitmap and stroke character sets that are defined in software and are portable. For example, we can put out a bitmap character that is 8 × 13 pixels by

```
glutBitmapCharacter(GLUT_BITMAP_8_BY_13, c)
```

where c is the number of the ASCII character that we wish to be placed on the display. The character is placed at the present **raster position** or

is part of the graphics state), is measured in pixels, and can be altered by the various forms of the function `glRasterPos*`. We return to text in Chapter 3. There we will see that both stroke and raster text can be supported most efficiently through display lists.

2.4.5 Curved Objects

The primitives in our basic set have all been defined through vertices. With the exception of the point type, all consist of line segments or use line segments to define the boundary of a region that can be filled with a solid color or a pattern. We can take two approaches to creating a richer set of objects.

First, we can use the primitives that we have to approximate curves and surfaces. For example, if we want a circle, we can use a regular polygon of n sides. Likewise, we have approximated a sphere with triangles and quadrilaterals. More generally, we approximate a curved surface by a mesh of convex polygons—a **tessellation**—which can occur either at the rendering stage or within the user program.

a. What is tessellation ?

ANSWER:

(Angel pp. 58)

approximate a curved surface by a mesh of convex polygons—a **tessellation**

b. What are other methods that can be used to represent curved objects?

ANSWER:

(Angel pp. 58)

The other approach, which we will explore in Chapter 11, is to start with the mathematical definitions of curved objects, and then build graphics functions to implement those objects. Objects such as quadric surfaces and parametric polynomial curves and surfaces are well understood mathematically, and we can specify them through sets of vertices. For example, we can define a sphere by its center and a point on its surface, or we can define a cubic polynomial curve by four points.

c. What methods are available in openGL?

ANSWER:

(Angel pp. 58)

Most graphics systems give us aspects of both approaches. In OpenGL, we can use the GLU or GLUT libraries for a collection of approximations to common curved surfaces, and we can write functions to define more of our own. We can also use the advanced features of OpenGL to work with parametric polynomial curves and surfaces.

2.4.6 Attributes

In a modern graphics system, there is a distinction between the type of a primitive and how that primitive is rendered. A red solid line and a green dashed line are the same geometric type, but they are rendered differently. An **attribute** is any property that determines how a geometric primitive is to be rendered. Color is an obvious attribute, as are the thickness of a line and the pattern used to fill a polygon. Several of these attributes are shown in Figure 2.19 for lines and polygons.

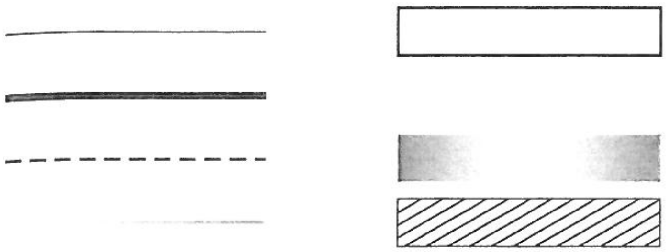


FIGURE 2.19 Attributes for (a) lines and (b) polygons.

a. What is a display list used for ?

ANSWER:

(Angel pp. 58)

Attributes may be associated with, or **bound** to, primitives at various points in the modeling and rendering pipeline. At this point, we are concerned with immediate-mode graphics. In **immediate mode**, primitives are not stored in the system but rather passed through the system for possible rendering as soon as they are defined. The current values of attributes are part of the state of the graphics system. When a primitive is rendered, the current attributes for that type are used, and it is displayed immediately. There is no memory of the primitive in the system. Only the primitive's effect on the frame buffer appears on the display; once erased from the display, it is lost. In Chapter 3, we introduce display lists, which will enable us to keep objects in memory so that these objects can be redisplayed.

b. What is a scene graph ?

ANSWER:

(Angel pp. 59)

OpenGL's emphasis on immediate-mode graphics and a pipeline architecture works well for interactive applications, but that emphasis is a fundamental difference between OpenGL and object-oriented systems in which objects can be created and recalled as desired. In Chapter 10, we will discuss scene graphs, which are fundamental to systems such as Java3D, and we will see that they provide another higher-level approach to computer graphics.

c. What are the attributes for a point , a line, a filed , text primitives ?

ANSWER:

(Angel pp. 59)

Each geometric type has a set of attributes. For example, a point has a color attribute and a size attribute. Line segments can have color, thickness, and pattern (solid, dashed, or dotted). Filled primitives, such as polygons, have more attributes because we must use multiple parameters to specify how the fill should be done. We can fill with a solid color or a pattern. We can decide not to fill the polygon and to display only its edges. If we fill the polygon, we might also display the edges in a color different from that of the interior.

In systems that support stroke text as a primitive, there is a variety of attributes. Some of these attributes are demonstrated in Figure 2.20; they include the direction of the text string, the path followed by successive characters in the string, the height and width of the characters, the font, and the style (bold, italic, underlined).

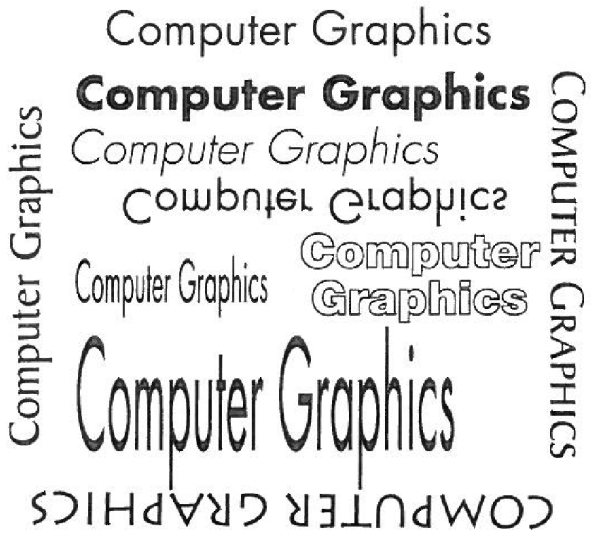


FIGURE 2.20 Stroke-text attributes.

2.5 COLOR

Color is one of the most interesting aspects of both human perception and computer graphics. We can use the model of the human visual system from Chapter 1 to obtain a simple but useful color model. Full exploitation of the capabilities of the human visual system using computer graphics requires a far deeper understanding of the human anatomy, physiology, and psychophysics. We will present a more sophisticated development in Chapter 7.

A visible color can be characterized by a function $C(\lambda)$ that occupies wavelengths from about 350 to 780 nm, as shown in Figure 2.21. The value for a given wavelength λ in the visible spectrum gives the intensity of that wavelength in the color.

Although this characterization is accurate in terms of a physical color whose properties we can measure, it does not take into account how we *perceive* color. As noted in Chapter 1, the human visual system has three types of cones responsible for color vision. Hence, our brains do not receive the entire distribution $C(\lambda)$ for a given color but rather three values—the **tristimulus values**—that are the responses of the three types of cones to the color. This reduction of a color to three values leads to the **basic tenet of three-color theory**: *If two colors produce the same tristimulus values, then they are visually indistinguishable.*

A consequence of this tenet is that, in principle, a display needs only three primary colors to produce the three tristimulus values needed for a human observer. We vary the intensity of each primary to produce a color as we saw for the CRT in Chapter 1. The CRT is one example of **additive color** where the primary colors add together to give the perceived color. Other examples of additive color include projectors and slide (positive) film. In such systems, the primaries are usually red, green, and blue. With additive color, primaries add light to an initially black display, yielding the desired color.

For processes such as commercial printing and painting, a **subtractive color model** is more appropriate. Here we start with a white surface, such as a sheet of paper. Colored pigments remove color components from light that is striking the surface. If we assume that white light hits the surface, a particular point will be red if all components of the incoming light are absorbed by the surface except for wavelengths in the red part of the spectrum, which are reflected. In subtractive systems, the primaries are usually the **complementary colors**: cyan, magenta, and yellow (CMY; Figure 2.22). We will not explore subtractive color here. You need to know only that an RGB additive system has a dual with a CMY subtractive system (see Exercise 2.8).

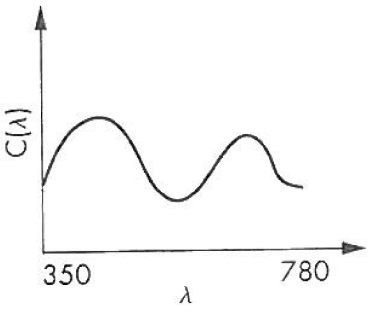


FIGURE 2.21 A color distribution.

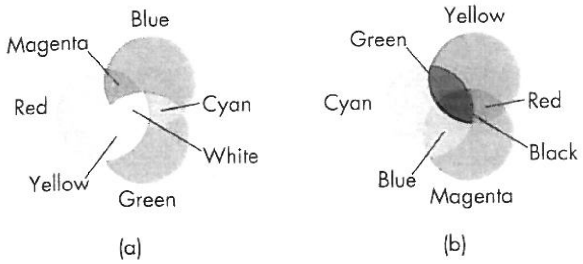


FIGURE 2.22 Color formation. (a) Additive color. (b) Subtractive color.

We can view a color as a point in a **color solid**, as shown in Figure 2.23 and in Color Plate 21. We draw the solid using a coordinate system corresponding to the three primaries. The distance along a coordinate axis represents the amount of the corresponding primary in the color. If we normalize the maximum value of each primary to be 1, then we can represent any color that we can produce with this set of primaries as a point in a unit cube. The vertices of the cube correspond to black (no primaries on); red, green, and blue (one primary fully on); the pairs of primaries, cyan (green and blue fully on), magenta (red and blue fully on), and yellow (red and green fully on); and white (all primaries fully on). The principal diagonal of the cube connects the origin (black) with white. All colors along this line have equal tristimulus values and appear as shades of gray.

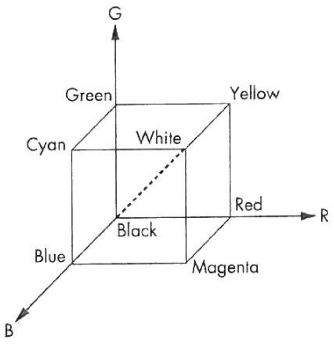


FIGURE 2.23 Color solid.

a. How do we get cyan in openGL ?

ANSWER:

(Angel pp. 61)
(0.0, 1.0, 1.0)

2.5.1 RGB Color

2.5.1 RGB Color

Now we can look at how color is handled in a graphics system from the programmer's perspective—that is, through the API. There are two different approaches. We will stress the **RGB-color model** because an understanding of it will be crucial for our later discussion of shading. Historically, the **indexed-color model** (Section 2.5.2) was easier to support in hardware because of its lower memory requirements and the limited colors available on displays, but in modern systems RGB color has become the norm.

In a three-primary-color, additive-color RGB system, there are conceptually separate buffers for red, green, and blue images. Each pixel has separate red, green, and blue components that correspond to locations in memory (Figure 2.24). In a typical system, there might be a 1280×1024 array of pixels, and each pixel might consist of 24 bits (3 bytes): 1 byte for each of red, green, and blue. With present commodity graphics cards having from 128MB to 512MB of memory, there is no longer a problem of storing and displaying the contents of the frame buffer at video rates.

As programmers, we would like to be able to specify any color that can be stored in the frame buffer. For our 24-bit example, there are 2^{24} possible colors, sometimes referred to as 16M colors, where M denotes 1024^2. Other systems may have as many as 12 (or more) bits per color or as few as 4 bits per color. Because our API should be independent of the particulars of the hardware, we would like to specify a color independently of the number of bits in the frame buffer and to let the drivers and hardware match our specification as closely as possible to the available display. A natural technique is to use the color cube and to specify color components as numbers between 0.0 and 1.0, where 1.0 denotes the maximum (or **saturated** value) of the corresponding primary, and 0.0 denotes a zero value of that primary. In OpenGL, we use the color cube as follows. To draw in red, we issue the following function call:

```
glColor3f(1.0, 0.0, 0.0);
```

a. What is alpha in openGL ?

ANSWER:

(Angel pp. 63)

Later, we shall be interested in a four-color (RGBA) system. The fourth color (A, or **alpha**) also is stored in the frame buffer as are the RGB values; it can be set with four-dimensional versions of the color functions. In Chapter 7, we will see various uses for alpha, such as for creating fog effects or combining images. Here we need to specify the alpha value as part of the initialization of an OpenGL program. If blending is enabled (Chapter 8), then the alpha value will be treated by OpenGL as either an **opacity** or **transparency** value. Transparency and opacity are complements of each other. An opaque object passes no light through it; a transparent object passes all light. Opacity values can range from fully transparent (A=0.0) to fully opaque (A=1.0).

b. What is glClearColor(1.0, 1.0, 1.0, 1.0) in openGL ?

ANSWER:

(Angel pp. 63)

defines an RGB-color clearing color that is white, because the first three components are set to 1.0, and is opaque, because the alpha component is 1.0. We can then use the function `glClear` to make the window on the screen solid and white. Note that by default blending is not enabled. Consequently, the alpha value can be set in `glClearColor` to a value other than 1.0 and the default window will still be opaque.

2.5.2 Indexed Color

2.5.2 Indexed Color

Early graphics systems had frame buffers that were limited in depth. For example, we might have had a frame buffer with a spatial resolution of 1280 × 1024, but each pixel was only 8 bits deep. We could divide each pixel's 8 bits into smaller groups of bits and assign red, green, and blue to each. Although this technique was adequate in a few applications, it usually did not give us enough flexibility with color assignment. Indexed color provided a solution that allowed applications to display a wide range of colors as long as the application did not need more colors than could be referenced by a pixel. This technique is still used today.

We follow an analogy with an artist who paints in oils. The oil painter can produce an almost infinite number of colors by mixing together a limited number of pigments from tubes. We say that the painter has a potentially large color **palette**. At any one time, however, perhaps due to a limited number of brushes, the painter uses only a few colors. In this fashion, she can create an image that, although it contains a small number of colors, expresses her choices because she can select the few colors from a large palette.

Returning to the computer model, we can argue that if we can choose for each application a limited number of colors from a large selection (our palette), we should be able to create good-quality images most of the time

a. What is color-lookup table in openGL ?

ANSWER:

(Angel pp. 63)

We can select colors by interpreting our limited-depth pixels as indices into a table of colors rather than as color values. Suppose that our frame buffer has k bits per pixel. Each pixel value or index is an integer between 0 and $2^k - 1$. Suppose that we can display colors with a precision of m bits; that is, we can choose from 2^m reds, 2^m greens, and 2^m blues. Hence, we can produce any of 2^{3m} colors on the display, but the frame buffer can specify only 2^k of them. We handle the specification through a user-defined **color-lookup table** that is of size $2^k \times 3m$ (Figure 2.25). The user program fills the 2^k entries (rows) of the table with the desired colors, using m bits for each of red, green, and blue. Once the user has constructed the table, she can specify a color by its index, which points to the appropriate entry in the color-lookup table (Figure 2.26). For $k = m = 8$, a common configuration, she can choose 256 out of 16 M colors. The 256 entries in the table constitute the user's color palette.

If we are in color-index mode, the present color is selected by a function such a

Input	Red	Green	Blue
0	0	0	0
1	$2^m - 1$	0	0
.	0	$2^m - 1$	0
.	.	.	.
.	.	.	.
$2^k - 1$	.	.	.
	m bits	m bits	m bits

FIGURE 2.25 Color-lookup table.

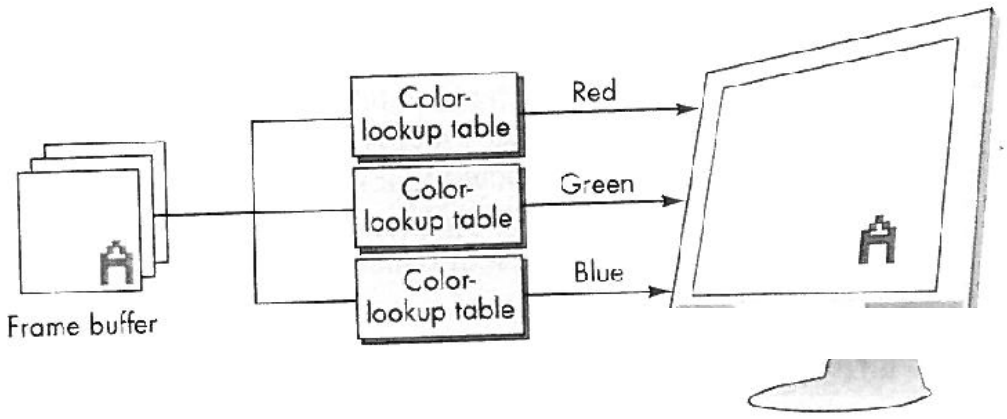


FIGURE 2.26 Indexed color.

2.10.3 Hidden – Surface Removal

Contrast this order to the way that we would see the triangles if we were to construct the three-dimensional Sierpinski gasket out of small solid tetrahedra. We would see only those faces of tetrahedra that were in front of all other faces as seen by a viewer. Figure 2.40 shows a simplified version of this **hidden-surface** problem. From the viewer's position, quadrilateral A is seen clearly, but triangle B is blocked from view, and triangle C is only partially visible. Without going into the details

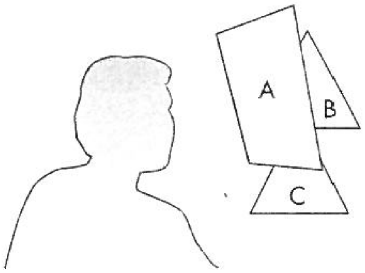


FIGURE 2.40 The hidden-surface problem.

of any specific algorithm, you should be able to convince yourself that given the position of the viewer and the triangles, we should be able to draw the triangles such that the correct image is obtained. Algorithms for ordering objects so that they are drawn correctly are called **visible-surface algorithms** or **hidden-surface–removal algorithms**, depending on how we look at the problem. We discuss such algorithms in detail in Chapters 4 and 7.

a. What is z-buffer algorithm ?

ANSWER:

(Angel pp. 84)

A hidden surface removal algorithm.

b. How is it requested and initialized in openGL ?

ANSWER:

(Angel pp. 84)

For now, we can simply use a particular hidden-surface–removal algorithm, called the **z-buffer** algorithm, that is supported by OpenGL. This algorithm can be turned on (enabled) and off (disabled) easily. In our main program, we must request the auxiliary storage, a z (or depth) buffer, by modifying the initialization of the display mode to the following:

```
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
```

Note that the z-buffer is one of the buffers that make up the frame buffer. We enable the algorithm by the function call

```
glEnable(GL_DEPTH_TEST)
```

either in `main.c` or in an initialization function such as `myinit.c`. Because the algorithm stores information in the depth buffer, we must clear this buffer whenever we wish to redraw the display; thus, we modify the clear procedure in the display function:

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

2.6 VIEWING

We can now put a variety of graphical information into our world, and we can describe how we would like these objects to appear, but we do not yet have a method for specifying exactly which of these objects should appear on the screen. Just as what we record in a photograph depends on where we point the camera and what lens we use, we have to make similar viewing decisions in our program.

A fundamental concept that emerges from the synthetic-camera model that we introduced in Chapter 1 is that the specification of the objects in our scene is completely independent of our specification of the camera. Once we have specified both the scene and the camera, we can compose an image. The camera forms an image by exposing the film, whereas the computer system forms an image by carrying out a sequence of operations in its pipeline. The application program needs to worry only about the specification of the parameters for the objects and the camera, just as the casual photographer is concerned about the resulting picture, not about how the shutter works or the details of the photochemical interaction of film with light.

There are default viewing conditions in computer image formation that are similar to the settings on a basic camera with a fixed lens. However, a camera that has a fixed lens and sits in a fixed location forces us to distort our world to take a picture. We can create pictures of elephants only if we place the animals sufficiently far from the camera, or we can photograph ants only if we put the insects relatively close to the lens. We prefer to have the flexibility to change the lens to make it easier to form an image of a collection of objects. The same is true when we use our graphics system.

2.6.1 The Orthographic View

See Part B of this Homework.

2.6.2 Two-Dimensional Viewing

See Part B of this Homework.

2.6.3 Matrix Modes

See Part B of this Homework.

2.7 CONTROL FUNCTIONS

We are almost done with our first program, but we must discuss the minimal interactions with the window and operating systems. If we look at the details for a specific environment, such as the X Window System on a Linux platform or Windows on a PC, we see that the programmer's interface between the graphics system and the operating and window systems can be complex. Exploitation of the possibilities open to the application programmer requires knowledge specific to these systems. In addition, the details can be different for two different environments, and discussing these differences will do little to enhance our understanding of computer graphics.

Rather than deal with these issues in detail, we look at a minimal set of operations that must take place from the perspective of the graphics application program. Earlier we discussed the OpenGL Utility Toolkit (GLUT); it is a library of functions that provides a simple interface between the systems. Details specific to the underlying windowing or operating system are inside the implementation, rather than being part of its API. Operationally, we add another library to our standard library search path. Both here and in Chapter 3, GLUT will help us to ur
characterize modern interactive graphics systems, including a wide range of APIs, operating systems, and window systems. The application programs that we produce using GLUT should run under multiple window systems.

2.7.1 Interaction with the Windows System

See Part B of this Homework

2.7.2 Aspect Ratio and Viewports

See Part B of this Homework

2.7.3 The *main*, *display*, and *myinit* Functions

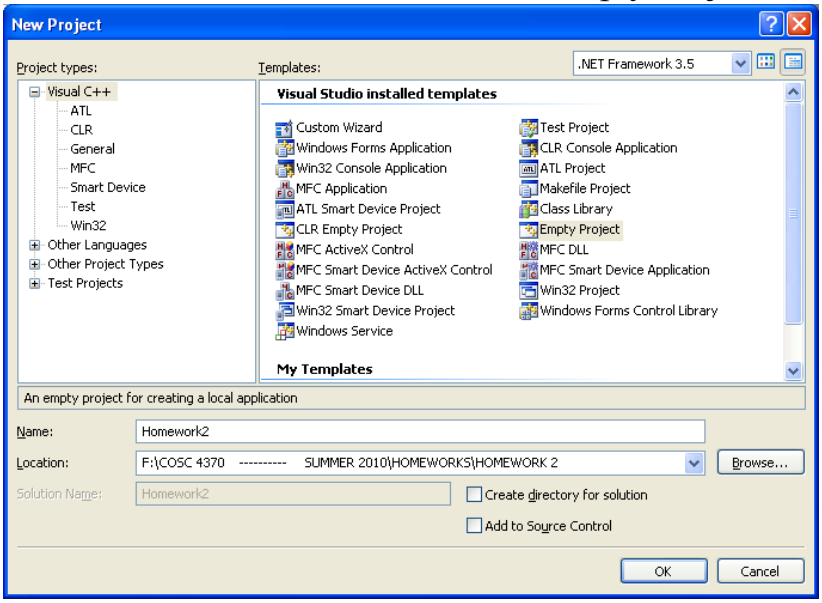
See Part B of this Homework

2.7.4 The Program Structure

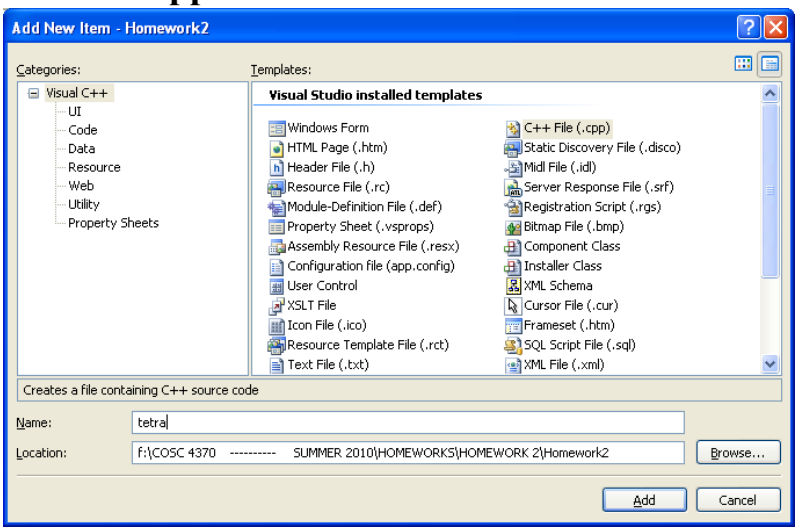
See Part B of this Homework

B. (100 pts) Visual Studio 2008 C++ Project

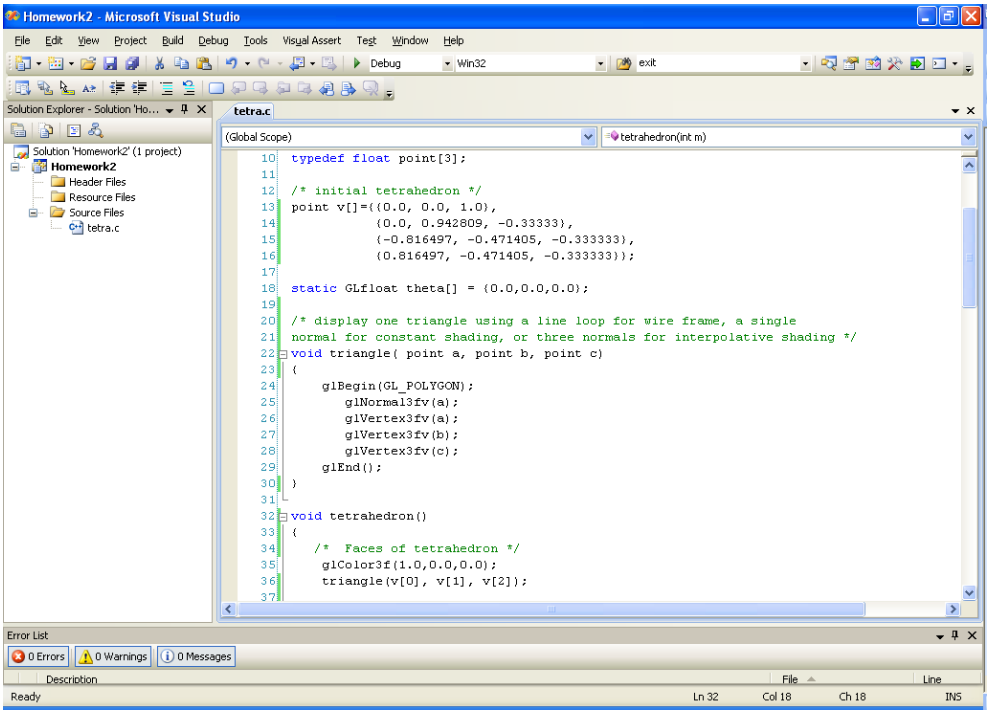
B1. Create Visual Studio 2008 C++, Empty Project, Homework2:



Create a .cpp file tetra.c



with the content:



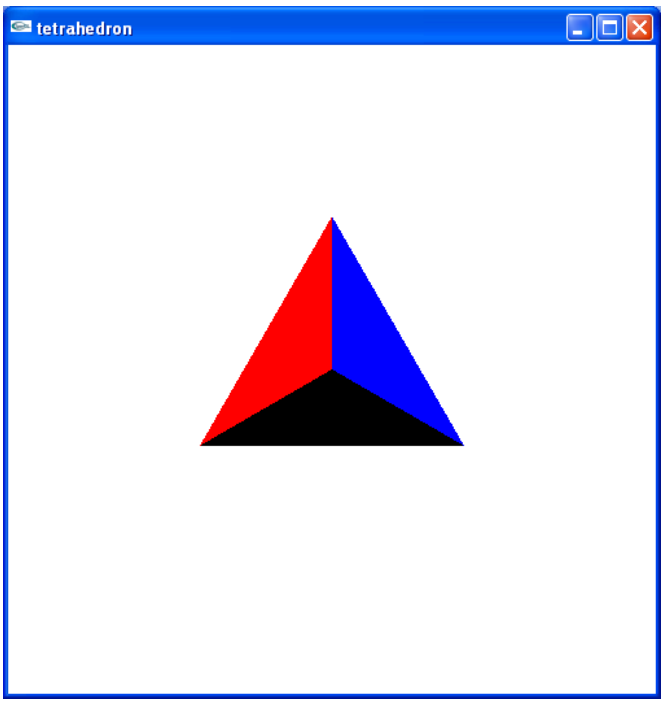
tetra.c

```
/* tetra.c */

#include <stdlib.h>
#ifdef __APPLE_
#include <GLUT/glut.h>
#else
#include <GL/glut.h>
#endif

typedef float vertex[3];

/* initial tetrahedron */
vertex vertices[]={
            {0.0,         0.0,        1.0},
            {0.0,         0.942809, -0.33333},
            {-0.816497, -0.471405, -0.333333},
            {0.816497,   -0.471405, -0.333333}};

static GLfloat theta[] = {0.0,0.0,0.0};

/* display one triangle using a line loop for wire frame, a single
normal for constant shading, or three normals for interpolative shading
*/
void triangle( int a, int b, int c)
{
    glBegin(GL_POLYGON);
       glNormal3fv(vertices[a]);

       glVertex3fv(vertices[a]);
```

```
       glVertex3fv(vertices[b]);
       glVertex3fv(vertices[c]);
    glEnd();
}

void tetrahedron()
{
   /*  Faces of tetrahedron */
      glColor3f(1.0,0.0,0.0);   // red
    triangle(0, 1, 2);

      glColor3f(0.0,1.0,0.0);   // green
    triangle(3, 2, 1);

      glColor3f(0.0,0.0,1.0);   // blue
    triangle(0, 3, 1);

      glColor3f(0.0,0.0,0.0);   // black
    triangle(0, 2, 3);
}

void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
      glLoadIdentity();
    tetrahedron();
    glFlush();
}


void myReshape(int w, int h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
        glOrtho(-2.0, 2.0, -2.0 * (GLfloat) h / (GLfloat) w,
            2.0 * (GLfloat) h / (GLfloat) w, -10.0, 10.0);
    else
        glOrtho(-2.0 * (GLfloat) w / (GLfloat) h,
            2.0 * (GLfloat) w / (GLfloat) h, -2.0, 2.0, -10.0, 10.0);
    glMatrixMode(GL_MODELVIEW);
    glutPostRedisplay();
}

void  main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(500, 500);
    glutCreateWindow("tetrahedron");
    glutReshapeFunc(myReshape);
    glutDisplayFunc(display);
    glEnable(GL_DEPTH_TEST);
    glClearColor (1.0, 1.0, 1.0, 1.0);
    glutMainLoop();
}
```

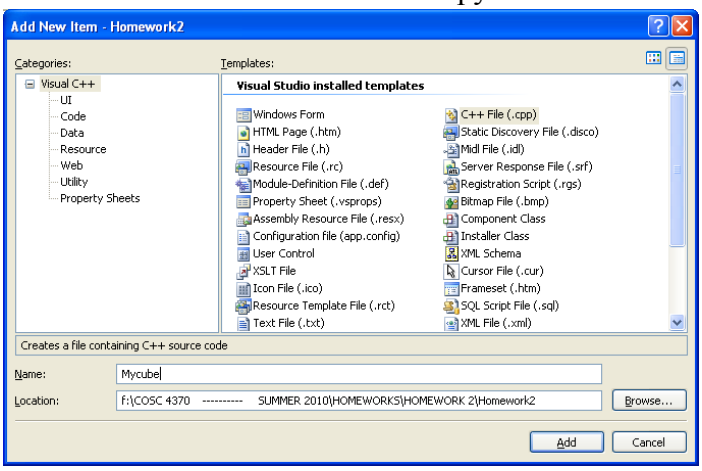
Build and run this Project: Insert a screenshot of your output.

ANSWER:



B2. The default viewing volume in openGL is a cube of side length 2 centered at (0,0,0). Thus, objects have to be inside this volume in order to be displayed:

Create new file **MYcube.c** and copy tetra.c into it.



After each modification, build the project to isolate faults.

1. Modify CreateWindow from "tetrahedron" to "MYcube".
2. Modify tetrahedron **function call** to MYcube.
3. Build MYcube just like the one above except that it is bounded by (-0.5, 0.5) in x, y, and z. (thus the corner **(-1,-1,-1)** becomes **(-0.5, -0.5, -0.5)**). vertices vertex[] will have 8 vertices. Please use the vertices as specified below:

4. Each face of the cube will be created by calling a polygon function instead of triangle (glBegin is GL_POLYGON) and passing to it the vertices that make up that polygon in the following order. Please place a comment of which face you are drawing, and use the following colors:

front (`glColor3f(1.0,0.0,0.0);` `//red`**),**
back (`glColor3f(0.0,1.0,0.0);` `//green`**),**
top (`glColor3f(0.0,0.75388,1.0);` `//blue`**),**
bottom (`glColor3f(1.0,0.2,0.7);` `//deep pink`**),**
right side (`glColor3f(1.0,1.0,0.0);` `// yellow`**),**
left side (`glColor3f(1.0,0.75,0.0);` `// orange`**).**

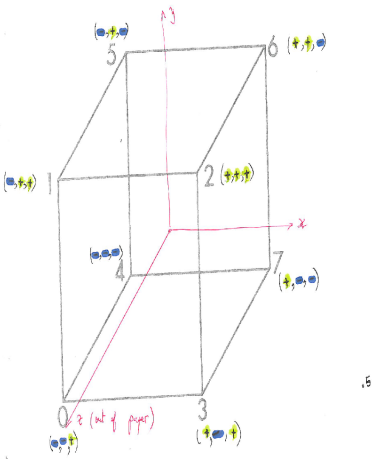
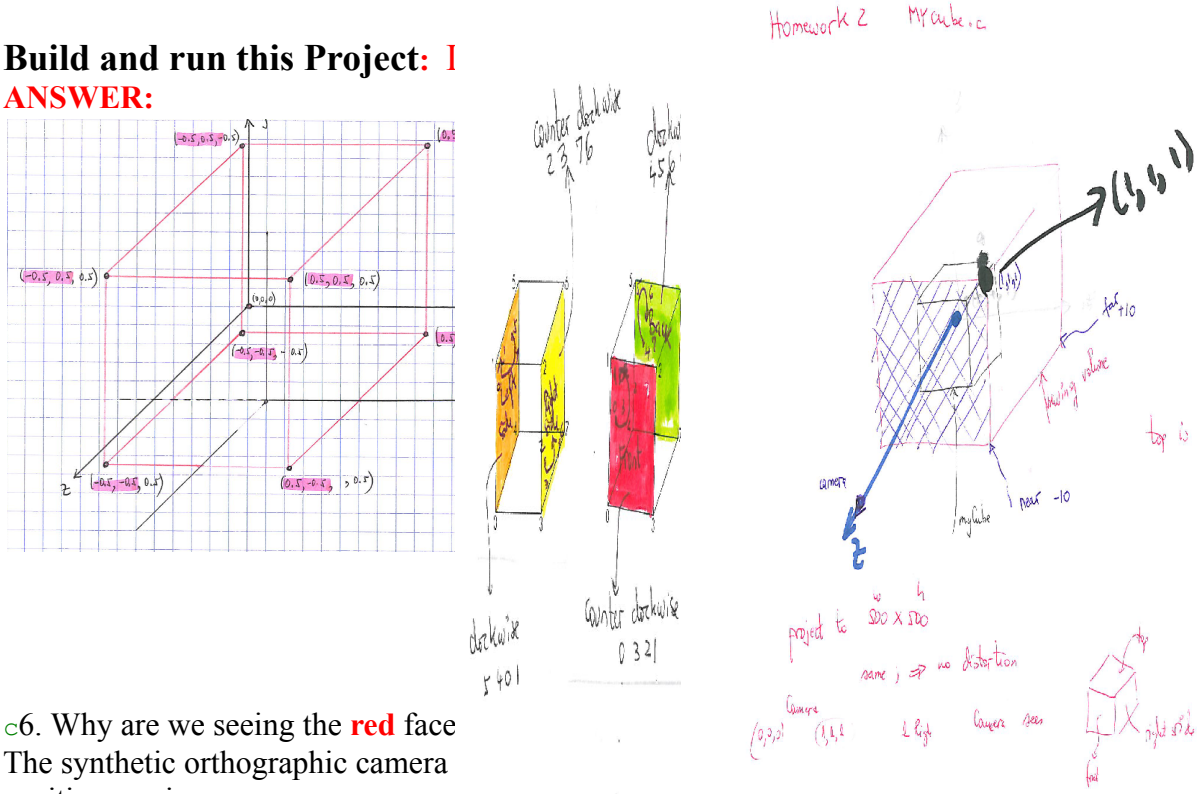


Figure 4.7 Numbering of cube vertices

5. Each face of the cube will be created by calling a face function instead of triangle. By definition, we signify a **front of the face** by **specifying the order of the vertices counter clockwise** and the **back of the face** by **specifying the order of the vertices clockwise.**
Thus, **Front face** is specified as 0,3,2,1, **Back face** is specified as 4,5,6,7, **Right Side face** is specified as 2,3,7,6 **etc.**

Build and run this Project: I
ANSWER:

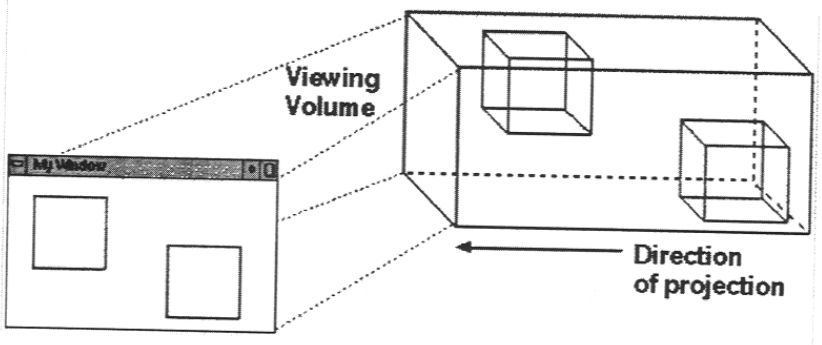


c6. Why are we seeing the **red** face
The synthetic orthographic camera positive z axis.

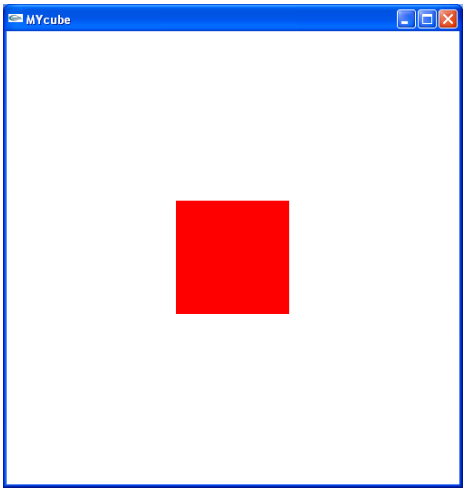
Orthographic Projections

```
GLvoid glOrtho( GLdouble left, GLdouble right,
                GLdouble bottom, GLdouble top,
                GLdouble nearClip, GLdouble farClip )
```

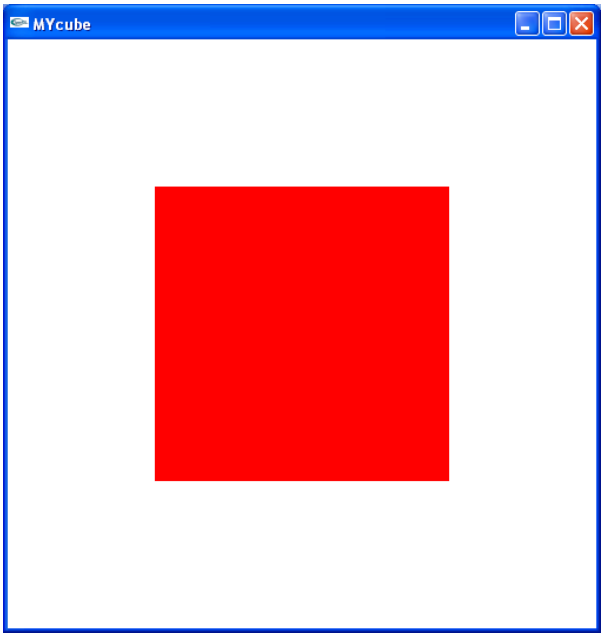
- Imitates parallel light rays approaching the eye
  - Also called a *parallel projection*
- Objects are projected to the screen by making them uniformly flat



Notes:



to display it like this:



When you use modeling transformation commands to emulate viewing transformations, you're trying to move the viewpoint in a desired way while keeping the objects in the world stationary. Since the viewpoint is initially located at the origin and since objects are often most easily constructed there as well (see Figure 3-9), in general you have to perform some transformation so that the objects can be viewed. Note that, as shown in the figure, the camera initially points down the negative z-axis. (You're seeing the back of the camera.)

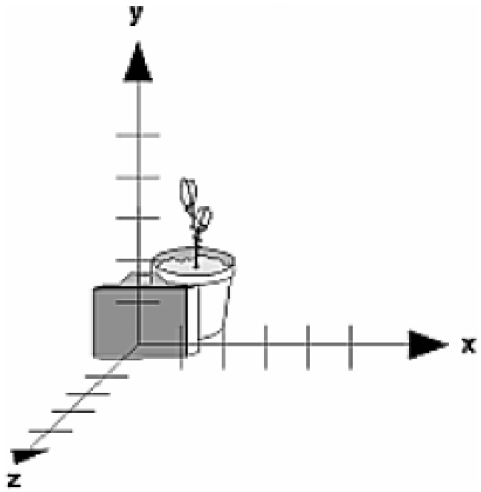


Figure 3-9 : Object and Viewpoint at the Origin

In the simplest case, you can move the viewpoint backward, away from the objects; this has the same effect as moving the objects forward, or away from the viewpoint. Remember that by default forward is down the negative z-axis; if you rotate the viewpoint, forward has a different meaning. So, to put 5 units of distance between the viewpoint and the objects by moving the viewpoint, as shown in Figure 3-10, use

```
glTranslatef(0.0, 0.0, -5.0);
```

This routine moves the objects in the scene -5 units along the *z* axis. This is also equivalent to moving the camera +5 units along the *z* axis.

Now suppose you want to view the objects from the side. Should you issue a rotate command before or after the translate command? If you're thinking in terms of a grand, fixed coordinate system, first imagine both the object and the camera at the origin. You could rotate the object first and then move it away from the camera so that the desired side is visible. Since you know that with the fixed coordinate system approach, commands have to be issued in the opposite order in which they should take effect, you know that you need to write the translate command first in your code and follow it with the rotate command.

Now let's use the local coordinate system approach. In this case, think about moving the object and its local coordinate system away from the origin; then, the rotate command is carried out using the now-translated coordinate system. With this approach, commands are issued in the order in which they're applied, so once again the translate command comes first. Thus, the sequence of transformation commands to produce the desired result is

```
glTranslatef(0.0, 0.0, -5.0);
glRotatef(90.0, 0.0, 1.0, 0.0);
```

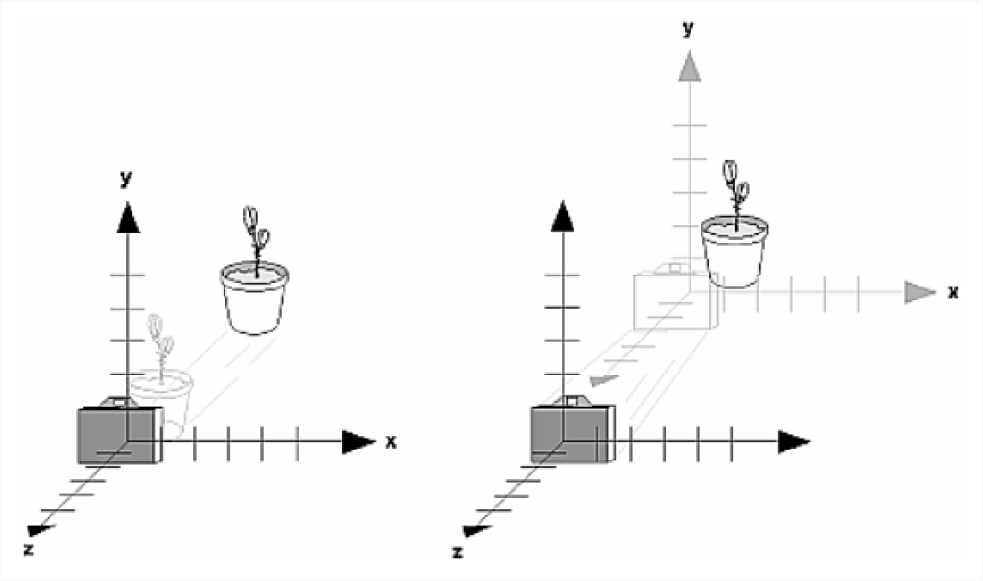


Figure 3-10 : Separating the Viewpoint and the Object

With an orthographic projection, the viewing volume is a rectangular parallelepiped, or more informally, a box (see Figure 3-15). Unlike perspective projection, the size of the viewing volume doesn't change from one end to the other, so distance from the camera doesn't affect how large an object appears. This type of projection is used for applications such as creating architectural blueprints and computer-aided design, where it's crucial to maintain the actual sizes of objects and angles between them as they're projected.

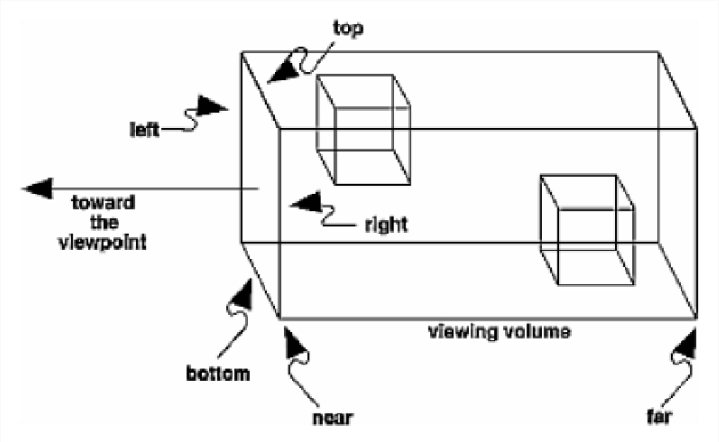


Figure 3-15 : Orthographic Viewing Volume

The command **glOrtho()** creates an orthographic parallel viewing volume. As with **glFrustum()**, you specify the corners of the near clipping plane and the distance to the far clipping plane.

*void **glOrtho**(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far);*

> *Creates a matrix for an orthographic parallel viewing volume and multiplies the current matrix by it. (left, bottom, -near) and (right, top, -near) are points on the near clipping plane that are mapped to the lower-left and upper-right corners of the viewport window, respectively. (left, bottom, -far) and (right, top, -far) are points on the far clipping plane that are mapped to the same respective corners of the viewport. Both near and far can be positive or negative.*

With no other transformations, the direction of projection is parallel to the z-axis, and the viewpoint faces toward the negative z-axis. Note that this means that the values passed in for far and near are used as negative z values if these planes are in front of the viewpoint, and positive if they're behind the viewpoint.

For the special case of projecting a two-dimensional image onto a two-dimensional screen, use the Utility Library routine **gluOrtho2D()**. This routine is identical to the three-dimensional version, **glOrtho()**, except that all the z coordinates for objects in the scene are assumed to lie between -1.0 and 1.0. If you're drawing two-dimensional objects using the two-dimensional vertex commands, all the z coordinates are zero; thus, none of the objects are clipped because of their z values.

*void **gluOrtho2D**(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top);*

> *Creates a matrix for projecting two-dimensional coordinates onto the screen and multiplies the current projection matrix by it. The clipping region is a rectangle with the lower-left corner at (left, bottom) and the upper-right corner at (right, top).*

The default clipping volume is a cube of side length 2 centered at the origine.

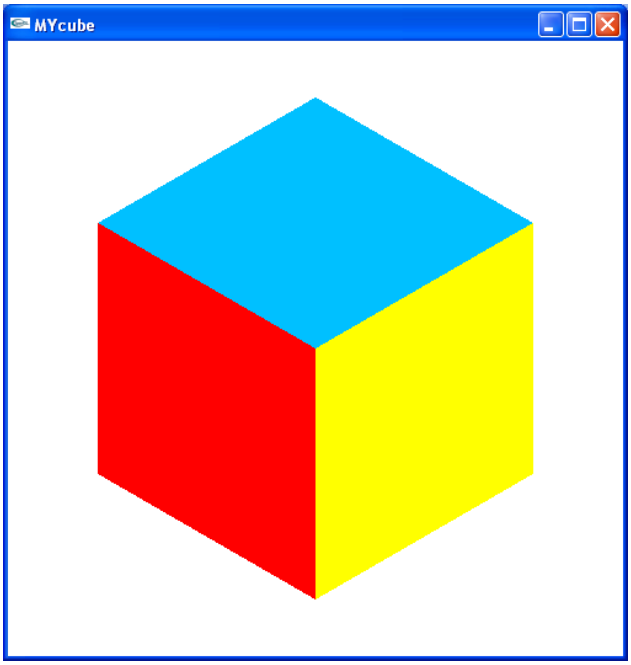
The clipping volume set in glOrtho() is measured from the origin in eye space. Thus, the near distance should be less than the far distance.

Moving the camera (eye) moves the near and far planes relative to the objects.

Build and run this Project: Insert a screenshot of your output.
ANSWER:

```
glOrtho(-1.0, 1.0, -1.0 * (GLfloat) h / (GLfloat) w,
            1.0 * (GLfloat) h / (GLfloat) w, -10.0, 10.0);
```

8. Adjust the camera so you can get the isometric image of MYcube below (**gluLookAt** added to display function). This means that camera needs to be placed on the line from (0,0,0) to (1,1,1), aiming at the cube's origin (0,0,0) and the up vector is on the y axis (0,**1**,0):



Consider the sequence illustrated in Figure 5.13. In part (a), we have the initial configuration. A vertex specified at **p** has the same representation in both frames. In part (b), we have changed the model-view matrix to **C** by a sequence of transformations. The two frames are no longer the same, although **C** contains the information to move from the camera frame to the object frame or, equivalently, contains the information that moves the camera away from its initial position at the origin of the object frame. A vertex specified at **q** through `glVertex`, *after* the change to the model-view matrix, is at **q** in the object frame. However, its position in the camera frame is **Cq**

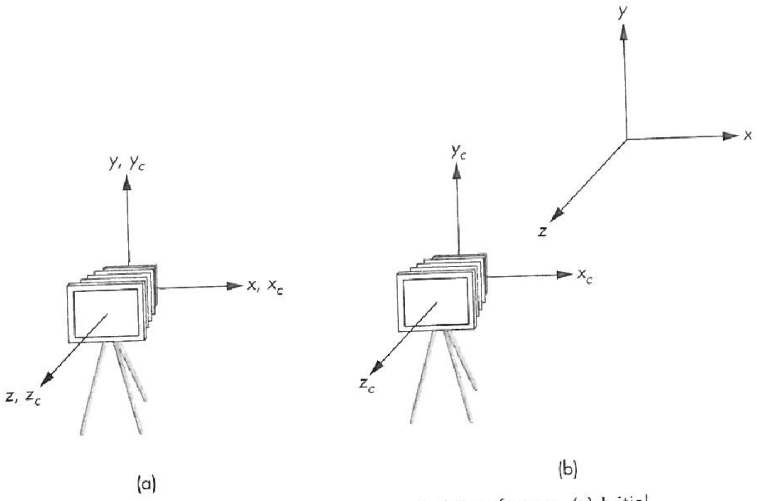


FIGURE 5.13 Movement of the camera and object frames. (a) Initial configuration. (b) Configuration after change in the model-view matrix.

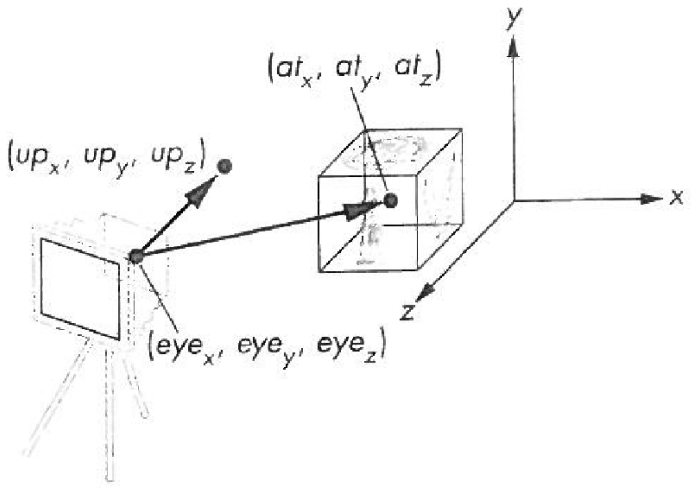


FIGURE 5.18 Look-at positioning.

Using the gluLookAt() Utility Routine

Often, programmers construct a scene around the origin or some other convenient location, then they want to look at it from an arbitrary point to get a good view of it. As its name suggests, the **gluLookAt()** utility routine is designed for just this purpose. It takes three sets of arguments, which specify the location of the viewpoint, define a reference point toward which the camera is aimed, and indicate which direction is up. Choose the viewpoint to yield the desired view of the scene. The reference point is typically somewhere in the middle of the scene. (If you've built your scene at the origin, the reference point is probably the origin.) It might be a little trickier to specify the correct up-vector. Again, if you've built some real-world scene at or around the origin and if you've been taking the positive y-axis to point upward, then that's your up-vector for **gluLookAt()**. However, if you're designing a flight simulator, up is the direction perpendicular to the plane's wings, from the plane toward the sky when the plane is right-side up on the ground.

The **gluLookAt()** routine is particularly useful when you want to pan across a landscape, for instance. With a viewing volume that's symmetric in both x and y, the (eyex, eyey, eyez) point specified is always in the center of the image on the screen, so you can use a series of commands to move this point slightly, thereby panning across the scene.

*void **gluLookAt**(GLdouble eyex, GLdouble eyey, GLdouble eyez, GLdouble centerx, GLdouble centery, GLdouble centerz, GLdouble upx, GLdouble upy, GLdouble upz);*

> *Defines a viewing matrix and multiplies it to the right of the current matrix. The desired viewpoint is specified by eyex, eyey, and eyez. The centerx, centery, and centerz arguments specify any point along the desired line of sight, but typically they're some point in the center of the scene being looked at. The upx, upy, and upz arguments indicate which direction is up (that is, the direction from the bottom to the top of the viewing volume).*

In the default position, the camera is at the origin, is looking down the negative *z*-axis, and has the positive *y*-axis as straight up. This is the same as calling

```
gluLookat (0.0, 0.0, 0.0, 0.0, 0.0, -100.0, 0.0, 1.0, 0.0);
```

The *z* value of the reference point is -100.0, but could be any negative *z*, because the line of sight will remain the same. In this case, you don't actually want to call **gluLookAt()**, because this is the default (see Figure 3-11) and you are already there! (The lines extending from the camera represent the viewing volume, which indicates its field of view.)

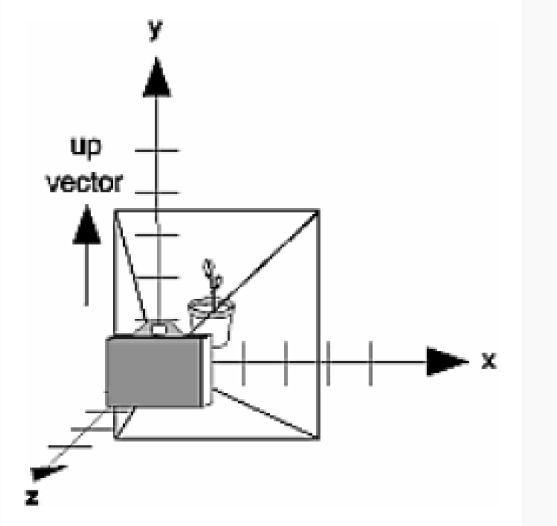


Figure 3-11 : Default Camera Position

Figure 3-12 shows the effect of a typical **gluLookAt()** routine. The camera position (eyex, eyey, eyez) is at (4, 2, 1). In this case, the camera is looking right at the model, so the reference point is at (2, 4, -3). An orientation vector of (2, 2, -1) is chosen to rotate the viewpoint to this 45-degree angle.

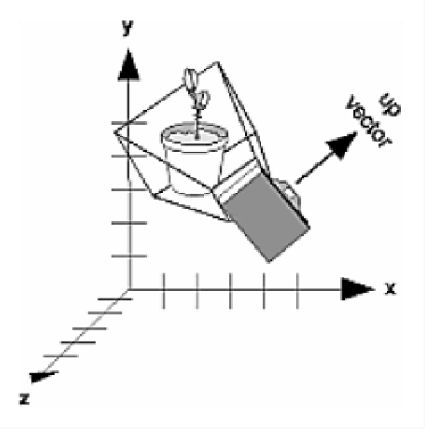


Figure 3-12 : Using gluLookAt()

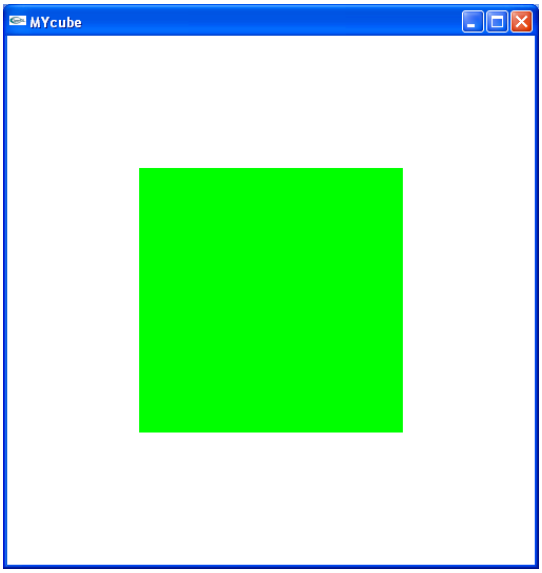
So, to achieve this effect, call

```
gluLookAt(4.0, 2.0, 1.0, 2.0, 4.0, -3.0, 2.0, 2.0, -1.0);
```

Build and run this Project: Insert a screenshot of your output.
ANSWER:

```
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    gluLookAt(1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
    MYcube();
    glFlush();
}
```

9. Adjust the camera so you can get the **Back** of MYcube below (**gluLookAt** added to display function). This means that camera needs to be placed on the line from (-1,0,0) looking at the origin of the cube (0,0,0) and up on the y axis (0,1,0):



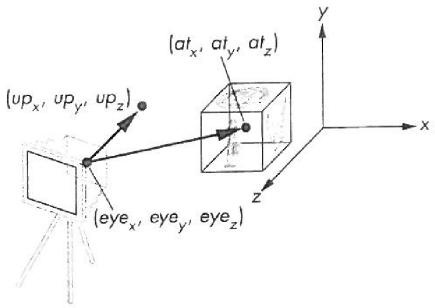
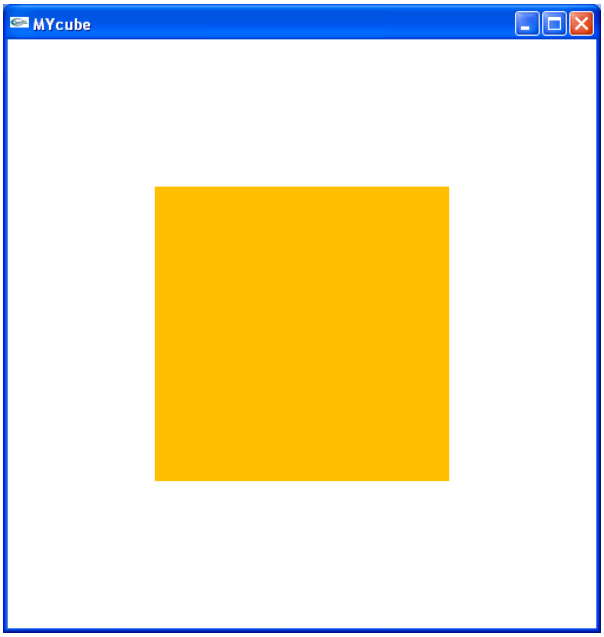


FIGURE 5.18 Look-at positioning.

Build and run this Project: Insert a screenshot of your output.
ANSWER:

```
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    gluLookAt(0.0, 0.0, -1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
    MYcube();
    glFlush();
}
```

10. Adjust the camera so you can get the **Left Side** of MYcube below (**gluLookAt** added to display function). This means that camera needs to be placed on the line from (-1,0,0) looking at the origin of the cube (0,0,0) and up on the y axis (0,1,0):



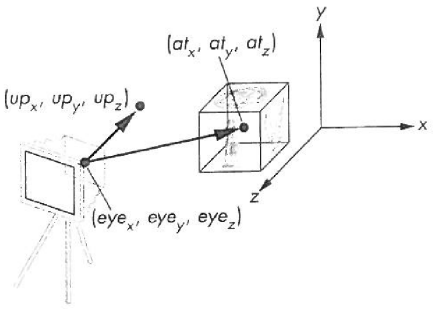


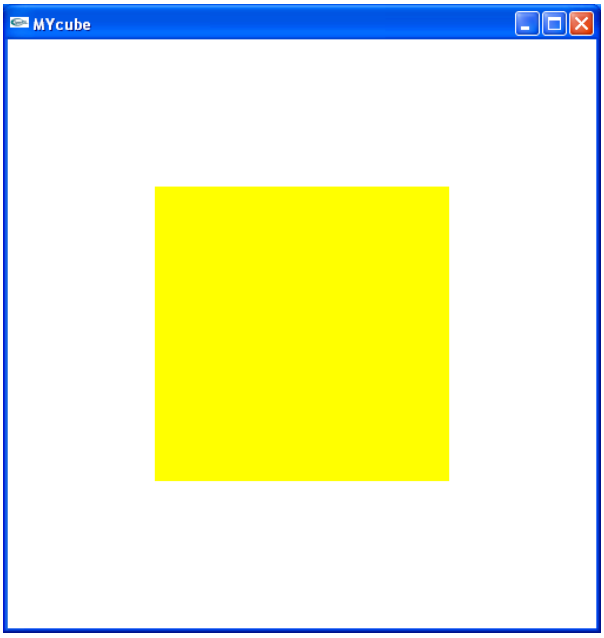
FIGURE 5.18 Look-at positioning.

Build and run this Project: Insert a screenshot of your output.
ANSWER:

```
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    gluLookAt(-1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
    MYcube();
    glFlush();
}
```

11. Adjust the camera so you can get the **Right Side** of MYcube below (**gluLookAt** added to display function). This means that camera needs to be placed on the line from (**?,?,?**) looking at the origin of the cube (0,0,0) and up on the y axis (0,1,0):

:



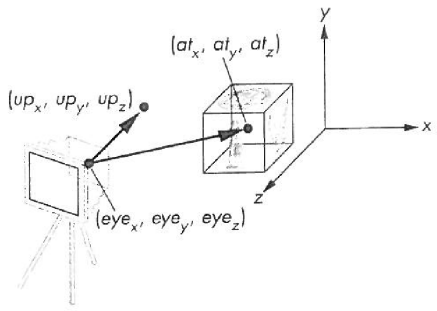
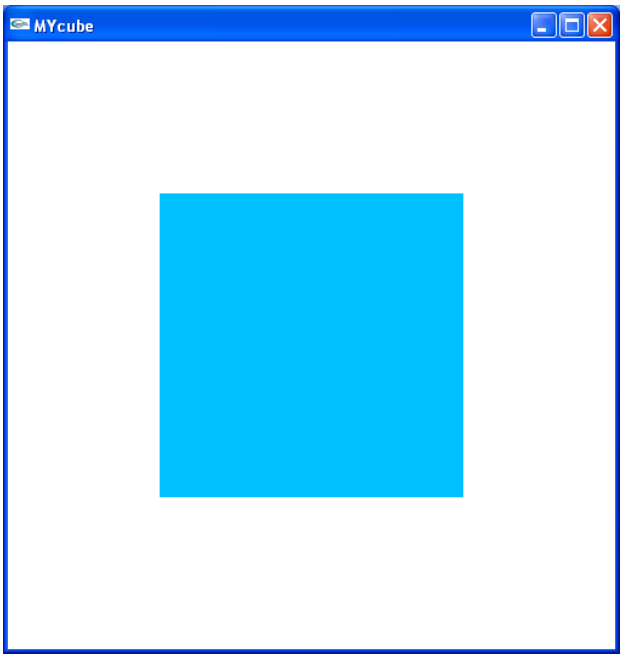


FIGURE 5.18 Look-at positioning.

Build and run this Project: Insert a screenshot of your output.
ANSWER:

```
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    gluLookAt(1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
    MYcube();
    glFlush();
}
```

12. Adjust the camera so you can get the **Top** of MYcube below (**gluLookAt** added to display function). This means that camera needs to be placed on the line from (**?**,**?**,**?**) looking at the origin of the cube (0,0,0) and up on the **?** axis (**?**,**?**,**?**):



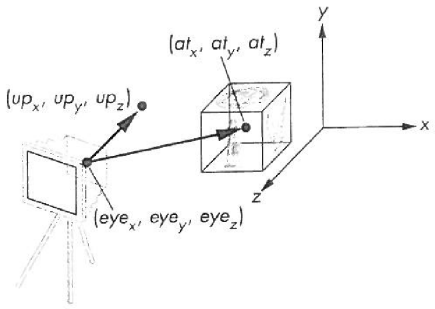
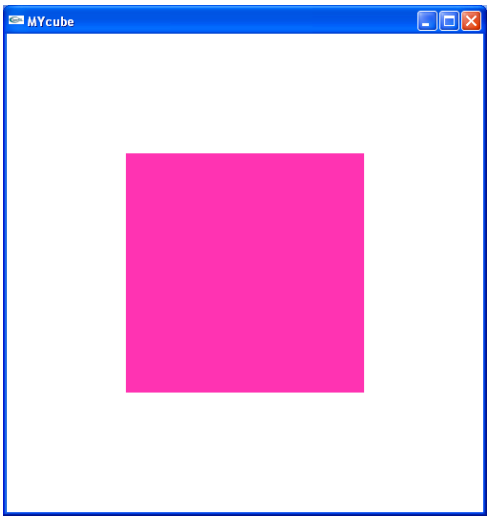


FIGURE 5.18 Look-at positioning.

Build and run this Project: Insert a screenshot of your output.
ANSWER:

```
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    gluLookAt(0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0);
    MYcube();
    glFlush();
}
```

13. Adjust the camera so you can get the **Bottom** of MYcube below (**gluLookAt** added to display function). This means that camera needs to be placed on the line from (**?**,**?**,**?**) looking at the origin of the cube (0,0,0) and up on the **?** axis (**?**,**?**,**?**):



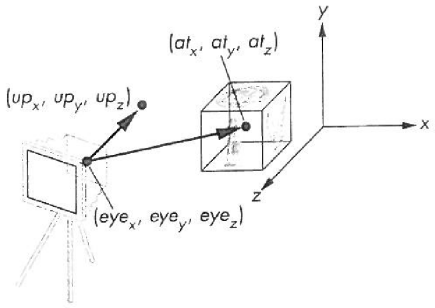


FIGURE 5.18 Look-at positioning.

Build and run this Project: Insert a screenshot of your output.
ANSWER:

```
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    gluLookAt(0.0, -1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0);
    MYcube();
    glFlush();
}
```